I agree with the conclusion that Michael H. Lutz and Phillip A. Laplante draw in their article “C# and the .NET Framework: Ready for Real Time?” (Jan./Feb. 2003): “C# and .NET might be appropriate for some soft and firm real-time systems” but “are not appropriate for hard real-time systems.” But it seems to me that more benchmarking work needs to be done to validate this conclusion.

The memory benchmark described in this article allocates and frees a linked list of 5,000 nodes in which the size of each node follows this progression: 2,500, 2,500, 5,000, 5,000, 7,500, 7,500, 10,000, 10,000, …, 57,500, 57,500, 60,000, 60,000.

Now, one of the things I’ve learned about benchmarking is that the conclusions drawn from a benchmark are only as strong as the benchmark’s design. So the question I have been asking myself after reading the article is “How well does this benchmark reflect real-time systems’ memory operations?”

This is, of course, an impossible question to answer, because no two real-time systems have exactly the same memory allocation behavior. In fact, some real-time systems allocate all their memory at the start and never free their memory until they shut down. Bearing that in mind, I have a few ruminations about the possible limitations of the benchmark presented.

I am not a real-time expert, but I understand that the “average program” does lots of small memory allocations (less than 100 bytes each) and comparatively few larger ones (more than 100 bytes). Also, most memory allocations are short-lived, and comparatively few are longer-lived. And, in fact, modern garbage-collecting memory systems are designed to take these behavior patterns into account (for example, “generational” garbage collection).

In contrast, the allocation sizes used in this benchmark are quite large compared to the “average program”—and they are all short-lived. So, this benchmark might produce apparent results that are quite different from real-world (or in this case, real-time) results.

For example, did you know that Microsoft .NET’s memory allocation system allocates objects that are larger than 20,000 bytes in their own separate heap? This allows them to be treated differently than objects in the normal heap (that is, memory is never compacted in the large memory heap). Consider that, in this benchmark, two-thirds of the memory allocations are over 20,000 bytes in size. Do you think this will have an effect on the memory allocation performance? It seems likely to me that it would—but it’s hard to say for sure without running a new set of benchmarks.

Another concern I have is that the direct comparison of the C memory allocator and
the .NET memory system is a tricky business. To some extent, we are comparing “apples and oranges” here. In particular, the .NET Framework zeroes all allocated memory before returning it, whereas C does not (assuming that this benchmark called malloc and not calloc). This will directly cause some of the nonlinear behavior shown in the graph (Figure 3 in the article), because allocating increasingly larger amounts of memory per object means zeroing more memory per allocation. Thus, much of the increasing time per object shown in Figure 3 might be due not to memory allocation and freeing per se but to the time spent zeroing the increasingly large memory blocks.

Thus, this benchmark tests the C library’s ability to allocate and free memory against the .NET Framework’s ability to allocate, free, and zero memory. This is not a fair comparison. To compare “apples and apples,” it would be interesting to run the C version of the test with calloc instead of malloc.

Now, you could argue that this test is “fair” from another point of view, since the .NET Framework does not have any analogue to malloc and is therefore inferior with respect to real-time systems. This might well be a reasonable argument. However, one counter to this argument could be that a .NET programmer could use P/Invoke to call a native allocation function (such as HeapAlloc) to perform nonzeroing allocations. The benefits of using the GC (for example, having to free the memory manually) would be lost, but this is no different than using malloc and thus provides an “oranges to oranges” comparison.

Finally, I would be remiss if I did not mention that the advantage of zeroing memory during allocation is in eliminating a class of difficult bugs involving uninitialized memory. This must be considered to be the value that is traded off against the increased performance provided by not zeroing memory.

In summary, I can’t say for sure whether further benchmarks would make .NET seem more or less appealing for real-time work, but someone with a deep knowledge of the average pattern of memory allocation of real-time systems and the .NET memory allocation system ought to run a revised set of carefully designed benchmarks and publish the results.

For what it’s worth, I am somewhat skeptical of .NET’s current applicability for real-time work, but I am cautiously optimistic about .NET in general. Hopefully, Microsoft will continue to listen to its developer customers—and give us all the wonderful software toys that every good engineer deserves on Christmas morning.

Adam Jackson
Software architect
adamjackson@aya.yale.edu
Michael Lutz and Phil Laplante respond:

In general, we agree with the observations that Mr. Jackson makes—and in particular, we agree that more work must be done in benchmarking C# and the .NET Framework. We also acknowledge that zeroing out memory is advantageous. But we feel the tests are not invalidated—rather, that the conclusion needs tweaking, as he has suggested, describing why the performance characteristics differ. Moreover, we should have noted that our conclusion is relevant only to those systems handling memory in a manner similar to our tests.

We would add to the points he makes that all benchmarks measure the results of particular tests only—nothing more, nothing less. So, even if we set up dozens of tests, there still might be a real-time system somewhere handling memory differently, and thus the tests might not apply to that system. Additionally, some real-time systems might not need dynamic memory allocation at all, particularly after initialization, and thus the whole issue is moot in such cases.

We emphasize that applicability of benchmarks must be considered on a case-by-case basis. For example, if a particular system handles memory differently at a fundamental level, it is necessary to run benchmarks specific to that approach. Additionally, machine characteristics, algorithm details, and potentially other variables (for instance, work in other process spaces) might all affect any benchmark’s relevancy to any particular system.

Our intent was not to discredit .NET. Indeed, we think Microsoft has done a good job designing and implementing the platform, and we were hoping the results would have been more positive. Nor did we intend this work to be a make-or-break decision maker regarding use of the .NET platform, but rather to provide one methodology and some results that could be considered in making such a decision. Nevertheless, we still think our conclusion is generally correct—that .NET requires additional tuning to match C from a memory management perspective. The performance difference still exists, independent of the cause.

Requirements Aren’t the Whole Picture—or Play

While Rob Austin and Lee Devin say some interesting things in “Beyond Requirements: Software Making as Art” (Jan./Feb. 2003), I don’t think their analogy of theater to software development proves anything about the place of formal requirements.
They do not make a convincing case that the two are more than superficially similar. Even if they used movie production instead of stage (since a movie, like software development, ends up with one shippable artifact), there is no obvious reason why the things that make a performance great are in any way analogous to the things that make a product great.

They talk about a consult-build-consult-build cycle. Why can’t that just as well be a consult-design-consult-design cycle, with the final design leading to requirements that guide implementation? Buildings and bridges, for instance, are designed iteratively but built only once, and that seems like a more natural analogy for software. The elements that lead to customer delight should be visible in the design without actually building them, whether the design is expressed as a prototype, paper-based screen flow mockups, or narrative descriptions. (“Design” here means external design—what the product looks like to the customer, not how it works inside.)

Also, some aspects of most software projects are not open to iterative discovery. Cellular telephones, for instance, must conform to air interface protocols that are fully and exhaustively specified. If my compiler vendor used an iterative process to decide which parts of the C standard to handle and how to interpret them, I would look for another vendor. Most products have some level of requirements that are not negotiable.

None of this is to diminish the promise of agile methods, which clearly are well suited to some kinds of projects. Although a lot of research remains to determine which agile practices matter most and the best way to combine them for different kinds of development, there is nothing surprising about iterative development with high customer involvement being a good idea. It’s just the analogy to theater that I don’t buy as meaningful.

Scott Preece
Distinguished Member of the Technical Staff
Motorola Urbana Design Center
preece@urbana.css.mot.com

Buildings and bridges, for instance, are designed iteratively but built only once, and that seems like a more natural analogy for software.